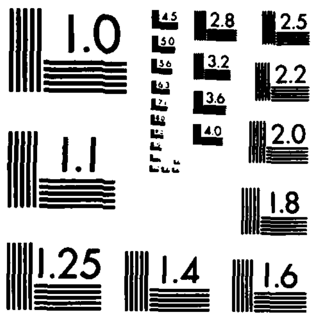
AD-A181 036 INTEGRITY AND THE FLEX PROGRAMMING SUPPORT ENVIRONMENT (U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY JUN 86 RSRE-MEMO-3915 1/1

UNCLASSIFIED DRIC-BR-101213 F/G 12/5 NL

END
DATE
FILMED
8-8?



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Royal Signals and Radar Establishment

Memorandum 3915

AUTHOR: Margaret Stanley.
DATE: June 1986

TITLE: Integrity and the FLEX Programming Support Environment

This paper discusses the integrity of the Flex Programming support environment, developed at RSRE, Malvern. It describes the special characteristics of the environment which result in a system of unusually high integrity. These characteristics include the capability mechanism, non-overwriting filestore and the use of procedures as first class objects.

Integrity and the FLEX Programming Support Environment

CONTENTS

Integrity and the FLEX Programming Support Environment

1. Introduction

Integrity is a vital factor in the acceptability of a Programming Support Environment (PSE). By integrity I mean the resistance of the support environment to corruption, from whatever cause, and the facilities provided to prevent illegal or unauthorised use of objects.

In this paper I discuss the integrity of the Flex PSE developed at RSRE Malvern. The fundamental mechanisms that enforce integrity include a capability mechanism, a non-overwriting filestore and the use of procedures as first class objects. I shall discuss each of these aspects indicating how they help in enforcing system integrity.

2. Aspects of Integrity

Users of a computer system expect that the system will preserve their data and their programs intact, coping with a hardware or a software failure without corruption of programs or data. Loss of data following a failure should be limited to loss of uncommitted changes. This is particularly important where large volumes of interrelated data are involved, such as in a database, where filestore corruption could result in massive loss. Users also expect that their own programs and data will not be accessible to other users without their authorisation. They expect protection not only when the system is used as specified, but also in the face of attempts to break the system and in the presence of incorrect programs. It should not be possible to compromise system integrity either accidentally or maliciously.

System integrity involves both mainstore integrity which is concerned with the running of programs and filestore integrity which is concerned with long term retention of values and of associations between values. Integrity is provided by a combination of computer architecture (hardware) complemented by features of the operating system and other software. The software has, where necessary and as far as possible, to supplement the architecture to prevent any unauthorised or improper use of mainstore or of filestore. Even hardware faults, software errors or tampering should not result in unauthorised access to values or in corruption of values. Unfortunately protection provided by software cannot be proof against tampering or against software errors.

2.1 Mainstore integrity

A mainstore value is part of one or more executing programs. It may be an invariant value (program code or constants) which must not be changed by any user or it may be program variables or workspace which must be protected from access by other programs or users. Integrity requires that mainstore values be used only in the proper way by programs which properly have access to them. Mainstore integrity can be compromised if a program can either read, execute or modify a part of mainstore to which the user has no rights because it is not properly in the current domain of his program or if a program makes wrong use of a legally accessed mainstore value, for example by changing a value which should be constant or by executing a data area. Mainstore integrity is also destroyed if any part of mainstore is released for re-use while it is still needed. If mainstore integrity is breached programs may behave in unanticipated ways or data may be illegally used or changed.

Illegal creation, alteration or use of mainstore addresses, sizes and controls on type of use (read, write or execute) must be prevented if the values to which the addresses give access are to be protected. Mainstore addresses and offsets that can be modified or created by program may be modified accidentally or maliciously giving illegal access to areas of mainstore belonging to other programs. A program such as a compiler may incorrectly calculate a mainstore address and point to the middle of a value, or it may pick up the wrong data to indicate the size of a value. If the data giving the size of a mainstore value can be modified by software there is no way to prevent a user who has access to a base address from reading a larger block of mainstore than he has a right to, thus gaining access to a contiguous value to which he may have no right. A user may then be able to read and even change mainstore values in the domain of another user's program.

The integrity of any system is weakened if special privilege can be invoked to by-pass specific integrity controls. For example partitioning mainstore into different user areas protects mainstore values against access by another user but the associated penalty is that partitioning prevents different users from sharing program code and constants in mainstore, nor does it allow procedures running in different user areas to communicate through shared areas in mainstore. The protection provided by partitioning is weakened if special privilege can be invoked to break the partition rules to enable code sharing between users of common utilities such as the editor. Similarly, if privileges can be obtained to override specific access or type of use controls the protection system is weakened.

Incorrect use (read, write or execute) of a mainstore value can compromise integrity. For example if a pointer gives access to the code of a procedure and the program interprets it as a program variable and updates it, the code will be corrupted with unpredictable results. Such corruption is difficult to detect and to recover from. Type of use may be controlled by data associated with a value or by partitioning mainstore so that values that can be updated are physically separated from code that can only be executed. However, the type of use data or the partition boundaries may themselves be held as data that can be modified. It is impossible to prevent tampering with values that can be set by software.

Mainstore garbage collection is provided by some compilation systems . It involves releasing for re-use any areas of mainstore that are no longer needed by programs that are running. It is necessary because executing programs run out of mainstore space but it can be a major hazard if mainstore addresses are not distinguishable from other values. If a value which is in fact a mainstore address is misinterpreted as a simple data value then the space to which it points may be released by the garbage collector for re-use, thus destroying the data value kept in that space. If the size information in the value is wrong the garbage collector may release part of the value. Such corruption can cause chaos.

2.2 Filestore integrity

Values on filestore (often called files) are values retained between user sessions. Files are often large values held as a sequence of non-contiguous blocks of filestore the ordering of which within a file may be indicated by chaining the blocks together as a linked list or by use of an index. Files are usually accessed through dictionaries or directories which are associations of names with files. Operating system software presents users with files rather than with blocks of filestore. A corrupted filestore is one in which the operating system presents the user either with no value for an existing file or (worse) with a wrong value. Integrity is breached either if any filestore corruption that can occur is not detected and recovered from or if a user can access or make unauthorised use of anything on filestore.

Possible reasons for filestore corruption include corruption of the association between a name and the sequence of filestore blocks; corruption of the sequencing information within the file; premature release of a filestore block currently in use in a file or corruption of the addresses and offsets used to locate the filestore blocks. If filestore corruption is detected there must be facilities to recover from it. However, detection of filestore corruption cannot be guaranteed. Operating system software cannot check the internal consistency of files and there may therefore be undetected loss of information. This is undesirable, and can be disastrous in critical applications.

Filestore integrity is breached if a system or software crash results in corruption of the filestore. One cause of filestore corruption is incremental updating of filestore. In incremental file updating a file consisting of several blocks is updated by a sequence of operations that overwrite existing blocks in the file and update the ordering of the blocks. If the system crashes when some but not all of these operations are complete the result will be an incompletely updated file perhaps with an inconsistent or a wrong index or linked list. Since the old file no longer exists because some blocks have already been changed, recovery is difficult. Simultaneous updating of a file by more than one user can also result in filestore corruption. This can be prevented by user or operating system software using semaphores and flags, but it is always possible to bypass software checks.

As with mainstore values, if filestore can be accessed using filestore addresses which can be created and manipulated by software there is nothing to prevent a user from by-passing the operating system controls to access the filestore directly using filestore addresses he has calculated for himself. He may even write a new value to a filestore block addressed by another user thus maliciously or accidentally modifying all or part of an existing file. It is very difficult to detect such unauthorised change or to recover from it. A vetting process that relies on enforcement by correct software in the operating system cannot be guaranteed.

Access and type of use controls are essential elements in integrity. Like mainstore values, files and parts of files must be used only in the proper way by users who are authorised to access them. Even privilege should not enable a user to access anything on filestore without the appropriate authorisation nor to use it in an unauthorised way.

Access to a file is usually provided through a directory in which it is named. A user gains access to his directories (his private environment) when he logs in. His private environment may be protected by passwords that are checked when he connects to the system. Individual files may also be protected by passwords checked by the operating system. If passwords (possibly encoded) are embedded somewhere in the operating system they can sometimes be decoded by a diligent hacker, particularly if he can create or modify mainstore or filestore addresses to gain unauthorised access to password tables.

An authorised user of a file should be unable to use it in an unauthorised way, such as writing to a file that has been declared to be read only or reading a file that has been declared to be execute only. The use (read,write or execute) to which a file may be put is often defined for different groups of authorised user such as owner only; a named group of users or any user. Operating system software checks that a user belongs to the appropriate group before permitting a particular type of use. If these controls are provided by a descriptor either attached to the file or by a textual or binary description of the access rules (not necessarily very firmly attached to the object) it is possible for a user (especially a privileged user) to change a file descriptor without necessarily being authorised (before the change) to access the file content.

Sharing of files should not lead to loss of control of the value by its original owner. A user may be able to authorise another user to read or to execute a file but after a user has permitted another user to share something by creating a new copy of it to his own directory, the original owner may lose control of the copied object. If the recipient of a shared file can modify the access rights on his copy, thus permitting, for example read access to a program that was issued with execute only access, it is easy to compromise the system. Conversely, if a file is shared without copying there must be protection to prevent the owner from deleting a shared file that is used by someone else. If not prevented, deletion of a shared file could either cause the sharer unexpectedly to lose the file or it could release a filestore block for re-use while still addressed from another directory.

Unauthorised access to files can occur if users are allowed to supply critical parameters to system procedures. For example, the user may be required to provide a parameter to a procedure for manipulating filestore, indicating which directory he is working on. If he provides an incorrect parameter the procedure may operate on the wrong directory or on a part of the filestore to which the user has no access rights. It should not be possible to supply as a procedure parameter a value that can cause the procedure to access areas of mainstore or of filestore to which the user would not normally have access.

3. What is FLEX ?

Flex is a multi-language Programming Support Environment (PSE) with a large amount of software available to users. It is built on the Flex capability object oriented architecture developed at RSRE, Malvern. The main design aim was to develop a system of high integrity and reliability and to simplify the development and maintenance of complex software. The result was a highly interactive PSE that is noticeably different from other PSEs. The PSE development since the first Flex architecture came into use in 1978 has been mainly a response to requests from programmers using the system. The software base includes all normal operating system facilities and many other procedures including compilers for Algol68 and Pascal. An Ada(*) compiler is near completion and an ML compiler is under development.

The Flex capability computer architecture[1,2] has (so far) been implemented in microcode on four hardware configurations including one multi-user implementation in which 3 Flex computers share a common filestore and common peripherals. The most recent implementation is on the ICL Perq.

A full description of Flex is beyond the scope of this paper, which will concentrate on those aspects that contribute to system integrity with

*Ada is a registered trademark of the US DoD.

some discussion of the effect this has on programmers and on the method of use of the Flex PSE.

4. Flex integrity features

The Flex computer has several features which enhance the integrity of the PSE. I shall describe these commenting on the effect they have on system integrity, with particular reference to the aspects detailed in section 2.

4.1 Capabilities

Flex is a capability machine. Those values in Flex which require some degree of access control to preserve system integrity are represented by capabilities. The capability mechanism ensures that values represented by capabilities can be used only in the way authorised by a capability and only by the holder of the appropriate capability. The Flex architecture provides the capabilities and enforces the mechanism, which is therefore not susceptible to interference from the software. The microcode (which is fixed and inaccessible to users, being used as an extension to the hardware) controls all access to values in mainstore and on backing store, permitting only legal operations to be performed. This ensures that only operations of the right kind are applied to mainstore and filestore values and only legally accessible values can be reached.

A capability can be created and modified only by the Flex microcode although capabilities can be treated like other values in that they can be held anywhere in mainstore or filestore and can be used in procedures. An important aspect of capabilities is that they are values that can be passed about quite freely in the same way as any other value such as an integer or a character string. There are mainstore capabilities that control access to mainstore objects (such as procedures), filestore capabilities that control access to objects on filestore and remote capabilities that control access to remote facilities (e.g. objects on other Flex computers). In a sense a capability is a pointer created on behalf of the user by the microcode, but the capability also holds the size of the value to which it gives access and contains information on the type of use (read only; read/write; execute) that will be permitted by the microcode.

The only way to use a facility on Flex or to get any access to mainstore, to filestore or to remote facilities is by using a capability. It is impossible to use software to create a capability either by manipulating other capabilities or store addresses or in any other way; a user can only get a capability by using an appropriate instruction (which issues it via the microcode). Mainstore addresses and filestore addresses have no meaning for a user, since the capability is not an address. It is rather a permission to use something (such as a region of store or a remote facility) in an authorised way. This means that users are unable to compromise the integrity of the system by unauthorised use of mainstore or filestore or by using store addresses.

Capabilities are more than pointers in that they also control the type of access that they provide (e.g. read only, execute only, read/write). The type of access provided is part of the capability. Since capabilities cannot be modified, the type of access permission that they allow cannot be changed. A user who wishes for a different type of access to an object must request a new capability, which can only be granted by the microcode. For example, a user with a read only capability to a block of mainstore cannot manipulate it to obtain read/write access. A new capability may give different access rights to the same value. Similarly, a user with a capability to execute a procedure can execute the procedure but he cannot do anything else to it, such as reading any internal values.

An example may clarify the concept of capabilities. Suppose that a user wishes to create a text file on filestore. He has a capability to execute the editor which gets a read/write capability for a region of mainstore, in which it holds the text being edited. When the editor is ready to write the text to filestore it requests a capability to create a read only block of appropriate size on filestore. It writes the text, using the create block capability, and gets back a read only capability for the block of text, which it passes to the user. This capability can be used only to read the text, and the text can only be accessed by using this capability. A user can pass the capability to another user, thus giving someone else read access to the text, or he can relinquish the capability if he no longer wishes to have access to the text. No other use can be made of that area of filestore as long as the capability exists, so the filestore cannot be over-written and then the old capability used to gain access to new data. The microcode takes care of this.

When the owner of a capability is willing to share the object to which it gives access, he gives the other user a copy of the appropriate capability. The object itself is not copied. Since capabilities cannot be tampered with, the recipient can use the shared object only in the authorised way.

The capability mechanism is a very powerful way of ensuring the integrity and security of the computing system. The fact that it is impossible for software to by-pass the microcode checks (which prevent unauthorised access to, or inappropriate use of, objects in mainstore or filestore) and that it is impossible to use software to forge or to modify capabilities or to retain them after the object to which they give access has ceased to exist, gives a degree of protection that is not available at this level on conventional computers. The protection is normally provided by the operating system software, which may usually be by-passed given sufficient knowledge.

4.2 Mainstore allocation

Mainstore allocation is handled by the Flex microcode and cannot be tampered with by a user. The store is allocated in disjoint blocks, which may be of any size, according to user requirements. The user is given a capability only for appropriate use of any block allocated to him (read only, read/write, execute) and the Flex system releases the block when it is no longer required.

The capability mechanism not only ensures that a block of store is correctly used but also that no user can gain access to any part of store unless he has an appropriate capability. A user cannot therefore corrupt store belonging to another user. Since the mainstore addresses, offsets and sizes are embedded in the capabilities and cannot be forged or altered by software, it is impossible for software to corrupt the store allocation. There is no concept on Flex of allocating a specific area of mainstore to any one user. A block of mainstore is allocated to a capability which may then be held by several users. Sharing objects such as system utilities is therefore easy and does not adversely affect the integrity of the mainstore. No privilege is required to make objects shareable.

When a user requires a larger block of mainstore than is available a rapid garbage collection is automatically invoked. User procedures (including utilities) are not involved in this process. The fact that any size of block can be allocated removes the difficulties and store waste associated with fixed size blocks. The fact that mainstore capabilities are easily distinguished from other types of mainstore values and that all mainstore allocation is controlled using capabilities enables the garbage collection to be undertaken without need for special software to check that the pointers have not been interfered with in any way.

4.3 Non-overwriting filestore

Filestore or backing store allocation is also dealt with by the Flex microcode. Filestore values may be of any size and can contain capabilities for other filestore values although they cannot contain mainstore capabilities. The fact that all filestore allocation is handled by the capability mechanism and that the filestore capabilities cannot be forged or tampered with by software gives a high degree of integrity to the filestore.

No user can delete a filestore value explicitly. He can merely release his copy of the capability for the value. The value itself cannot be erased while a capability for it exists in any other filestore value. This means that it is impossible to delete a filestore value shared by another user, for which the other user has a capability.

The Flex architecture is designed to prevent the possibility of partially updating filestore. Flex filestore is non-overwriting, apart from a few root pointers which are updated as a single atomic action. There is no capability to alter an existing filestore value (other than a root pointer) so a user cannot alter the content or size of an allocated block of filestore. He can either request the microcode to write something to filestore for him or he can read or execute something that has already been written. The updatable root pointers are single word blocks updated as a unitary operation. They usually contain a reference to a dictionary from which other allocated blocks can be reached. The dictionary contains a set of associations of names with filestore capabilities.

An apparent change to a filestore value involves getting a capability for the new (changed) value. The old (unchanged) value continues to exist until all copies of the capability for it have been deleted. For example, when an editable file is apparently changed by editing, as described earlier, what actually happens is that the editor writes the edited file to new filestore blocks and returns a new capability for the edited file. At this stage both the old and the new (changed) files exist as separate values. A user may have capabilities for both. He may choose to retain both capabilities or to delete his copy of the capability that he no longer wants. The old file has not changed and holders of a capability for it will still be able to access it. The user has simply gained access to another editable file. The new editable file, although written to filestore, will not be accessible in subsequent user sessions unless a root pointer is updated to point to a value, such as a dictionary, that gives access to it.

The architecture ensures that the root pointer update is unitary. Since the only operation that can overwrite anything on disc is unitary, partial overwriting is impossible. Successive values of a root pointer each give access to a completely consistent set of unchangeable filestore values. If the system crashes at any time the user will either have the old value of the root or the new value. He cannot have a partially updated value. Since partially updated filestore is impossible software to detect or recover from partial updates is unnecessary.

The unitary updating of root pointers is supported by a microcode check that removes from the user the responsibility of preventing two users from corrupting the filestore by updating the same value simultaneously. The microcode checks that when a user attempts to write a new value to a root pointer he knows the existing value. He will fail if another user has changed the value of the root pointer since he read it. If he fails he simply needs to read the new value of the root pointer and change the new dictionary instead of the old one. The only way to change a dictionary is to create a new one (which is an edited version of the old dictionary) and then update the root pointer to point to the new dictionary. If two users having parallel access to the same filestore value both attempt to update the same root pointer both will succeed. Unless both users are changing the association with the same name nothing will have been lost. However, if both users are changing the association with the same name, the earlier update will be lost, although the filestore will remain in a consistent state so there is no loss of integrity. The loss of an update could be controlled by software, using semaphores, if users were sufficiently concerned.

4.4 Sharing filestore capabilities

The Flex module (the unit of separate compilation on Flex) illustrates the flexibility of the access protection provided by the ability of a filestore value to contain other filestore capabilities. A Flex module is a filestore value containing both the capability to use some compiled code and a capability to deliver the source text from which it was derived. If the owner of the compiled code wishes to share the capability to use the code, while denying to others the ability to examine the source text he issues a capability for the module with the source text capability removed. This allows sharing of software while providing protection against unauthorised access to the source text of the software.

4.5 Procedures as first class objects

Flex treats procedures as first class, context independent values or objects[5,6]. A procedure is a true procedure value in the sense of Landin[4]. Procedure code needs a context (its non-locals) to make it executable and on Flex the context or environment in which the procedure runs is bound in as a part of the procedure value. The procedure value is a capability to execute the procedure. The possession of a procedure capability allows the holder only to execute the procedure. It does not allow him to dismember the procedure to find how it works, the values of its non-locals or what other procedures it might use. An important consequence of treating a procedure as a value like any other value is that procedure capabilities may be passed to other users, used as parameters for other procedures and even delivered by other procedures (provided the language also supports this notion, as does Algol68).

4.5.1 Delivered procedures

Delivering a procedure from a procedure means that values (both local and non-local) and input parameters of the delivering procedure can be bound into the delivered procedure and hidden from a caller of the delivered procedure. This facility is a useful means of information hiding and provides a safe way of hiding passwords and other access controls from prying. The facilities provided are not privileged. They can be used by any programmer.

For example, consider a procedure, make_channel, that creates a channel for passing messages and a pair of procedures for accessing that channel. (Each message consists of a vector of characters.) Make_channel takes an integer giving the size of channel (i.e. the number of messages it can hold). It delivers three channel access procedures, write_channel that writes a message into the channel, read_channel that reads a message from the channel and change_key that allows a user to lock the channel, hiding the new key. One channel is created by each call of make_channel, and that channel can be accessed only by using the procedures delivered by that call.

The Algol68 mode of procedure make_channel is:

```
PROC make_channel=(INT size)
                          STRUCT(PROC(VECTOR[]CHAR)VOID write_channel,
                                    PROC VECTOR [] CHAR read_channel,
                                    PROC VOID change_key);
```

The first delivered procedure (write_channel) takes a vector of characters and delivers a void. Each time it is executed it writes one message (a vector of characters) into the channel (taking action as defined in procedure make_channel to deal with a full channel or a busy channel).

The second delivered procedure (read_channel) takes no parameters and delivers a vector of characters. Each time it is executed it reads one message from the channel (taking action as defined in procedure make_channel to deal with an empty channel or a busy channel).

The third delivered procedure (change_key) allows a user to lock the channel. Change_key prompts the user for a key or password. The key that is supplied will be required by procedures change_key, read_channel and write_channel before they give access to the channel. Failure to respond with the correct key will cause the access procedures to fail.

The user who invokes make_channel need not know how the access procedures work. He can use them to access the channel and he can pass them to other users to enable them to access that channel. The users of the channel need not know the size of channel (input to procedure make_channel), nor how the procedures work. The channel is a non-local value of both read_channel and write_channel bound to these procedures when make_channel is executed. The key or password is a non_local of all three access procedures bound to them when make_channel is executed and changed only by change_key. It does not exist outside these procedures.

Similarly, a filestore capability can be hidden inside a set of access procedures, as in the example below.

```
PROC make_read_file= (INT file_capability)
                          STRUCT(PROC INT read_next_int,
                                 PROC BOOL read_next_bool,
                                 PROC BOOL is_empty);
```

The delivered procedures (all giving access to the same file parameter of make_read_file) may be stored on backing store because all the internal values and non-locals can be stored on backing store. Different procedures may be issued to different users, providing communication through the shared data structure. No user need know the structure of the stored data, nor need they have access to every value in the structure. The access procedures may perform any desired checks before giving access to the file.

4.5.2 Protection from diagnostics

When using procedures to hide values such as the channel and key in the above example it is clearly important that users of the procedures cannot look at the internal working to find the protected values. A procedure capability allows execution of the procedure but does not permit the user to read the code. The use of a debugger and dump facilities to investigate the reason for a procedure failure must not give access to values normally protected in a procedure. It is possible on Flex to prevent such illegal access. Following a failure (because of an explicit failure or as a result of an internal error such as dividing by zero, or an Ada exception) a value (called an exception value) will normally be returned to the calling procedure giving information on the cause of the failure and on the local values of the failed procedure that are currently in scope. Values bound in to a procedure as non-locals are automatically excluded from exception values. In addition the Flex architecture allows a programmer to protect sensitive procedures by preventing them from providing any information to the exception value. The calling procedure can handle the exception value internally or it can itself fail, adding information on its own values to the exception value that it passes to its own calling procedure. Exception values thus accumulate information as they are passed out through the calling procedures to the command interpreter. The information contained in an exception value will normally give the user access to local values in the failed procedure and to the source text from which the procedure was derived. If a module is being used from which the source text capability has been removed, an exception value cannot be used to gain access to source text. The exception value is the only value delivered by a failed procedure.

An exception value can be used by a diagnostic procedure to present to the user information on the state of the failed procedures at the point of failure. In the channel example given above, if any of the channel access procedures failed the exception value would give access to the local values of the procedure but not to the channel or to the key which are non-local values. The content of the channel can be discovered only by invoking read_channel and only the content of the channel will then be delivered.

4.5.3 Protection from unauthorised access

Since to use any object in Flex filestore or mainstore a user must have the appropriate capability, which cannot be forged or altered, it is clearly important that capabilities themselves be protected from theft by other users. Like other values, capabilities are protected by hiding the only copy of the capability within access procedures. The protected capability can then be shared safely by issuing the capabilities for the access procedures. The protected value cannot be reached except by executing an access procedure because the value is bound into the procedure and does not exist elsewhere. The access control provided by the capability mechanism combines with the protection provided by procedure values to give an unusually powerful and flexible form of access control that can be used, not only by the operating system, but also by any programmer, to protect values.

The access procedures are ordinary procedures created by a user. No privilege is required. Before allowing a user to reach a protected value, the access procedure may perform whatever checks it likes. It may require a complicated sequence of actions. It may even record for future analysis all attempts at unauthorised use. It will not necessarily request a password, although it may do so. If a password is requested it need not be a single word. When a protection check fails the access procedure fails, denying the user access to the protected values. Having successfully executed all protection checks, the access procedure allows the user to reach the protected value, perhaps by calling the command interpreter. The user will still be executing the access procedure. The value will again be hidden from the user on exit from the access procedure. The access procedure can be given to anyone in the knowledge that they still need to satisfy the built in checks before they can reach the protected value.

An example of the use of procedures on Flex to protect capabilities is found in the protection of each user's private environment (dictionaries of name/value associations). The capabilities for the dictionaries in the environment are embedded in a procedure called a user-id procedure. Access to the environment is granted only while the user-id procedure is running and only after the protection checks (such as passwords) have been satisfied. A user-id procedure cannot be invoked from within another user-id procedure because the environment set up within a user-id procedure does not include the names/values of the other user-id procedures. Invoking the user-id procedure is the Flex analogue of logging in, and exiting from the user-id procedure is the Flex analogue of logging out on a conventional system. An entire session on Flex therefore takes place during a single execution of the user-id procedure. The passwords do not exist outside the user-id procedure and so cannot be discovered. They can only be used.

4.5.4 Binding parameters to procedures

As illustrated above, the facility to deliver procedures from procedures allows critical parameters to be bound into procedures that operate on sensitive data. Critical parameters can therefore be hidden from the user, who cannot change them. Safeguards against misuse of system procedures by users are provided by binding the user dependent critical parameters, such as the user's dictionary, into the procedures delivered to a user. The user is therefore protected against calling such procedures with a wrong parameter.

For example, a procedure to modify a dictionary on Flex does not have the dictionary as a parameter. It is delivered to the user environment with the dictionary that it modifies bound into it. It cannot operate on another dictionary, or on any other filestore object. A user is thus prevented from accidentally or maliciously modifying a dictionary to which he has no right or from using a parameter to access someone else's values.

Another example illustrating the binding of parameters to prevent unauthorised change to shared values is provided by the way in which a module (compiled unit) is protected against change except by its creator. Every Flex user possesses a capability for an amend procedure that allows him to amend modules created by him with new compiled code derived from changed source text. The amend procedure is delivered to the user with the capability for the user's dictionary bound into it. Every module on Flex includes a capability for a procedure to assign new compiled code to the module. The assign procedure in a module has the dictionary of the creator of the module bound into it. In order to modify any specific module the user calls his amend procedure with the existing module and the changed code as parameters. The amendment will take place only if the dictionary bound into the user's amend procedure is the same as the dictionary bound into the module's assign procedure (i.e. only if the user also created the module).

A module issued to another user still contains its assign procedure capability but the recipient does not have acccss to the capability for the creator's amend procedure. The capability for a module can therefore be shared, giving use of the compiled code and access to the source text from which it was derived (unless that capability was excluded) while denying the ability to change the module. A shared module is thus protected against unauthorised change.

5. How good is the integrity of Flex?

The capability mechanism, the non-overwriting filestore, and the ability to treat procedure capabilities as first class objects combine to give a very robust system in which hardware or system failure or software abuse cannot result in a corrupted system. It is impossible, using software, to by-pass the safeguards against corruption provided by the Flex architecture.

Taking the points raised in section 2.

5.1 Mainstore integrity

Since all mainstore allocation on Flex is handled by capabilities that include not only the base address but also size and type of use information, and capabilities cannot be forged or modified using software it is not possible to corrupt the system by misuse of mainstore addresses or by using incorrect size and or type of use information. A user cannot accidentally or maliciously read or destroy values from another user's program.

It is not necessary to separate user areas nor to partition mainstore according to type of use. Mainstore capabilities belonging to a single user may be scattered throughout the mainstore. Sharing mainstore data values to provide communication between users or sharing procedure code when different users invoke the same procedure, can be done safely, with proper use of the values enforced by the capability mechanism. Privilege is not needed. The different users each hold a capability for the same shared object.

Mainstore garbage collection is safe and rapid because capabilities are easily distinguished from data values. It is impossible to confuse a mainstore pointer and a data area or to pick up the wrong size information from a capability.

Protection provided by mainstore capabilities is proof against tampering. There is no need to rely on operating system software to enforce either access to or correct use of mainstore.

5.2 Filestore integrity

The risk of corruption resulting from partially updated filestore objects and dictionaries after a hardware or system failure is avoided on Flex. Incremental updating of files is impossible because the filestore is non-overwriting with the exception of a few root values that can be updated only as unitary operation. Detection of and recovery from filestore corruption, always difficult and error prone, is therefore unnecessary.

Parallel updating of filestore values by more than one user on Flex cannot result in filestore corruption. If two users do access a filestore value in parallel and both attempt to change the same value in the same dictionary (by naming an object or by changing a module) both users will succeed but the last update to be committed will supersede the first. Although this can be frustrating since the changes made by the first user will apparently have been ignored, the integrity of the filestore is maintained. Some nugatory work may have been done but the resultant loss of information never leads to inconsistency. Loss of updates could be avoided by provision of software semaphores and flags to prevent parallel access.

Direct access to filestore using filestore addresses has no meaning on Flex. All filestore allocation is performed by the microcode which issues capabilities to use filestore. Users cannot address filestore except by using an issued capability in the authorised way. Since capabilities cannot be forged or modified by software, and since filestore addresses have no existence independent of the capabilities, no user, even with privilege, can access anything on filestore without the appropriate authorisation. Normal integer data cannot be confused with capabilities or with filestore addresses so a user cannot create a capability or modify one to write to a part of the filestore already occupied by a filestore value.

The type of use which may be made of a filestore value is part of the capability and cannot therefore be modified or ignored. Operating system software is not involved in enforcing correct use of values reached through capabilities. Sharing filestore objects is safe on Flex because even when a user has permitted another user to share something by issuing the appropriate capability or by placing the capability in a shared dictionary the original owner does not lose control of the shared object. The recipient cannot modify the access rights on his capability, for example to gain read access to a program that was issued with execute only access, nor can he modify a filestore value to which he has access rights. He can use the shared value only in the authorised way. A user who reads and updates a filestore value issued by another user (which he can do only if the capability gives him read access) has created a new value. He has not affected the value that was originally given to him.

Protection for a user's private environment (dictionaries of name/value associations) or for any other capability value is provided by embedding the capability in a procedure. Access is granted only while the procedure is being executed and only after all access checks have been satisfied. The access checks may involve passwords and can be as complicated as the user wishes. The checks are embedded with the protected item in the procedure and it is impossible for users to dismember any procedure to find out what checks have been imposed. The Flex architecture excludes non-local values hidden in a procedure from the exception value created when the procedure fails so diagnostic techniques cannot be used to give access to protected values. Programmers can also explicitly deny information to the exception value. Embedding values in procedures is not a privileged operation so programmers have complete flexibility in imposing their own access controls over and above the controls provided by the capability mechanism. This is much more secure than the usual access protection where passwords and type of use access controls are values only loosely associated with the protected filestore value and with the operating system software that checks them.

If a shared capability is protected within an access procedure the sharer is given the capability for the access procedure (with its built in access checks) rather than the capability for the value that is protected. The facility to include capabilities for filestore values within other filestore values allows flexible protection such as is given to the source text from which the compiled code in a shared module was derived. The capability to deliver source text need not be delivered to a user of the module to enable him to use the compiled code.

It is possible using procedure values to permit free use of critical operating system procedures that operate on dictionaries and other sensitive values without the risk that they will be abused by supplying them with wrong parameters. Programmers can embed critical values in procedures both to prevent unauthorised access and to prevent the use of wrong parameters where the correct use of parameters is crucial. This is achieved by delivering the holding procedure from another procedure which performs the necessary binding. In particular critical parameters to system procedures are embedded in the procedures supplied to users, so they cannot be changed. For example, system procedures which can affect a user environment by altering a dictionary, are available only within that environment, and have the user dictionary bound into them.

Protection to prevent unauthorised change to shared modules is provided by embedding dictionaries within procedures. Each user has an amend procedure with the capability for his own dictionary bound into it. Each module has an assign procedure with the capability for the dictionary of the creator of the module bound into it. A module can be amended only if these two dictionary capabilities are the same.

6. Conclusions

Flex is a PSE of unusally high integrity. The three main features that provide integrity are available to any user without privilege. They are capabilities, non-overwriting filestore and procedure values.

Every user of Flex uses the capability mechanism, since there is no other way to access values on Flex. This, together with the non-overwriting filestore gives confidence that the integrity of the filestore will not be breached by hardware failure or system crash and makes it unnecessary to provide special facilities to detect or to recover from filestore corruption or to enforce correct use of values in mainstore or in filestore.

Any programmer can protect his values, be they capabilities or other values, by embedding them inside procedures, providing his own procedures to check passwords. He can protect values from access through diagnostics by keeping them as non-local values in the procedures. He can share modules with other users and yet retain confidence that they cannot amend the shared compiled code and he can if he wishes deny a sharer access to the source text from which the shared compiled code was derived. Since any programmer can bind critical parameters into procedures, system procedures and other procedures can be issued for general use without the danger that they can be abused to gain illegal access to values by supplying a wrong parameter.

The improvement in software productivity resulting from use of a PSE with the integrity that is provided by Flex has not been measured, but the absence of many of the problems that beset programmers on conventional systems and the knowledge that it is unnecessary to provide special software to recover from filestore corrruption must result in improved productivity.

The integrity features of Flex just described are but a few of several unusual and useful features of the Flex PSE. Future work on Flex is aimed at making the PSE and the ideas it demonstrates more widely available, and at improving the facilities. The underlying capability architecture and the non-overwriting filestore are not expected to change but additional facilities are being worked on to extend the capability mechanism to work across networks of computers. The possibility of implementing Flex on existing computer systems (without re-microcoding) is being considered as a topic for future research.

7. References

1. "PerqFlex Firmware" by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Report 85015 December 1985

2. "Flex: A working computer with an architecture based on procedure values." by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Memorandum 3500. 1982.

3. "Curt: The command interpreter for Flex" by I.F.Currie and J.M.Foster. RSRE Memorandum 3522. 1983.

4. "The mechanical evaluation of expressions" by P.J.Landin Computer Journal Vol 6, No 4, pp308-320. Jan 1964.

5. "In praise of procedures" I.F.Currie RSRE Memorandum 1982

6. "Using true procedure values in a programming support environment" M.Stanley. RSRE Memorandum 3916, 1986

DOCUMENT CONTROL SHEET

Overall security classification of sheet ...UNCLASSIFIED...

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3915	3. Agency Reference	4. Report Security Classification U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

Integrity and the flex programming support environment

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Stanley M	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This paper discusses the integrity of the Flex Programming support environment, developed at RSRE, Malvern. It describes the special characteristics of the environment which result in a system of unusually high integrity. These characteristics include the capability mechanism, non-overwriting filestore and the use of procedures as first class objects.

MED

8